Splitting Polyhedra to Generate More Efficient Code

Harenome Razanajato, Vincent Loechner, Cédric Bastoul

Splitting Polyhedra to Generate More Efficient Code

Efficient Code Generation in the Polyhedral Model is Harder Than We Thought

Harenome Razanajato
University of Strasbourg,
CNRS, INRIA (France)
harenome.ranaivoarivony-razanajato@inria.fr

Vincent Loechner
University of Strasbourg,
CNRS, INRIA (France)
vincent.loechner@inria.fr

Cédric Bastoul
University of Strasbourg,
CNRS, INRIA (France)
cedric.bastoul@inria.fr

ABSTRACT

Code generation in the polyhedral model takes as input a union of $\mathbb{Z}$-polyhedra and produces code scanning all of them. Modern code generation tools are heavily relying on polyhedral operations to perform this task. However, these operations are typically provided by general-purpose polyhedral libraries that are not specifically designed to address the code generation problem. In particular, (unions of) polyhedra may be represented in various mathematically equivalent ways which may have different properties with respect to code generation. In this paper, we investigate this problem and try to find the best representation of polyhedra to generate efficient code.

We present two contributions. First we demonstrate that this problem has been largely under-estimated, showing significant control overhead deviations when using different representations of the same polyhedra. Second, we propose an improvement to the main algorithm of the state-of-the-art code generation tool CLooG. It generates code with fewer tests in the inner loops, and aims to reduce control overhead and to simplify vectorization for the compiler, at the cost of a larger code size. It is based on a smart splitting of the union of polyhedra while recursing on the dimensions. We implemented our algorithm in CLooG/PolyLib, and compared the performance and size of the generated code to the CLooG/isl version.

Keywords

Code generation, control overhead reduction, disjoint union of $\mathbb{Z}$-polyhedra, polyhedral model

1. INTRODUCTION

The polyhedral model is a theoretical framework for analyzing, transforming, and optimizing static control parts of programs, or affine loop nests. It is based on a geometrical representation of the iteration domain of each statement, as the intersection between a convex parametric polyhedron of any dimension and a lattice. Such a set of points is also known as a parametric $\mathbb{Z}$-polyhedron. The polyhedral model relies on three steps:

- raising the original code in the geometrical view, as a set of $\mathbb{Z}$-polyhedra associated to each statement;
- performing some geometrical transformations in this view;
- lowering back the set of $\mathbb{Z}$-polyhedra to generated code.

In this paper, we focus on this latter code generation phase and specifically on the problem of generating efficient code.

A naïve code generation algorithm consists in generating code scanning the convex hull of the union of $\mathbb{Z}$-polyhedra provided as input, and then protecting each statement in the core of the generated loop nest by a test, checking if this iteration is contained in the domain of this statement. It leads to the smallest code size, but with very poor runtime performance.

The state-of-the-art code generation algorithm was introduced by Quilleré, Rajopadhye and Wilde [12] and extended by various subsequent works to further improve the quality of the generated code [2, 3, 13, 6]. Quilleré et al.'s algorithm computes a disjoint union of $\mathbb{Z}$-polyhedra at each level of a recursion on the dimensions, and successively generates code for each resulting subset. It will generate larger output code, but with lower runtime complexity, since iteration domain membership does not have to be checked at runtime. On the other hand, some tests and multiple loop bounds leading to calls to *min* and *max* are not eliminated, since the splitting is done at a coarse granularity at each considered level of the recursion. CLooG is a popular implementation of that algorithm which includes various improvements to avoid code explosion, notably by reducing the complexity of the splitting, reducing in the same way the number of different scanned subsets, and the size of the generated code, without degrading performance[1].

We propose a new extension of CLooG's algorithm that eliminates multiple loop bounds and tests. In some cases, especially when they appear in the inner loops, the multiple loop bounds and the tests induce a high control overhead. They also sometimes prevent the compiler from generating efficient vectorized code. For example, when tiling a non-rectangular 2-dimensional domain, the inner loop trip count is not constant, and the compiler will guard the vectorized version by a runtime test checking the number of iterations of the loop.

IMPACT 2017
Seventh International Workshop on Polyhedral Compilation Techniques
January 2017, Stockholm, Sweden
In conjunction with HiPEAC 2017.

http://impact.gforge.inria.fr/impact2017

[1] http://www.cloog.org

Our proposal is based on the observation that the generated code depends on how $\mathbb{Z}$-polyhedra are represented. For instance, a given $\mathbb{Z}$-polyhedron may be represented by many mathematically equivalent unions of $\mathbb{Z}$-polyhedra. In code generation tools, the choice of a given representation is usually left under the responsibility of the underlying polyhedral library. But depending on that choice, the generated code may be different, including at the performance level. In this paper, we demonstrate the important impact of this choice and we propose an agressive splitting of the union of polyhedra, specifically designed to get a convenient polyhedral representation for the code generation problem. Our splitting strategy will specialize different cases in the code generation algorithm recursion. But everything comes with a price: our algorithm will generate larger code, and must sometimes be limited to handle only inner loops in order for the code size to remain within an acceptable limit. It is the usual code size versus performance dilemma.

This paper is organized as follows. Section 2 recalls the main algorithm of CLooG. Our proposal is depicted on a motivating example in Section 3, and then fully detailed in Section 4. Section 5 presents our results and a comparison to the standard CLooG/isl version on a set of benchmarks, taken from the CLooG distribution and from the PolyBench. Finally, Section 6 presents some related work and we conclude.

2. CLooG

Let us first present an overview of CLooG's main algorithm, as proposed by Bastoul [3].

CLooG takes as input a list of polyhedra, each one being associated to a program statement S_i; and a list of scheduling functions to be applied to each of them. These transformations can possibly be non-unimodular, non-invertible, non-integral or non-uniform. Applied to the integer points contained in their associated polyhedron, they define the list of $\mathbb{Z}$-polyhedra that the resulting code has to scan.

Each of these $\mathbb{Z}$-polyhedra is represented in a simpler manner, as the integer points in a polyhedron, thanks to the *generalized change of basis* technique proposed by Le Verge [10]: it is not necessary to compute the actual image of the original polyhedron integer points by the transformation, but it is sufficient to *add* the transformed dimensions to the original ones. Thus, the original integer points are kept, and we get the $\mathbb{Z}$-polyhedron of interest represented in a larger dimensional polyhedron also containing the original one.

CLooG's algorithm is based on a recursion along the dimensions, from outermost ($d = 1$) to innermost ($d = n$). To generate loop level d, it will first compute the projection of the polyhedra to the d outermost dimensions and separate them into an ordered list of disjoint polyhedra. Then it will scan this list of subdomains to generate code for loop level d and recurse in each loop for dimension $(d + 1)$ using the list of polyhedra touched by them. The detailed algorithm is given in Figure 1.

Step 5 of the algorithm scans the subdomains for loop level d. For each of these subdomains, it will compute the lower bound and the stride for this loop: those two values are computed from the inner polyhedra that will be scanned in the subdomain. In order to keep the code compact, it will avoid any separation of the inner polyhedra by merging them if possible at step 5(b). Then, at step 5(c), it will recursively call the function, giving as context the incoming context intersected with the bounds of the currently generated loop: the generated loop index will act as a parameter in the following recursive calls. Step 7 will merge back some *point* polyhedra that were separated from their neighboring higher dimensional *host* polyhedra, to reduce the generated code size.

In the next section, we will apply this algorithm on an example, and we will present our extension that eliminates costly multiple loop bounds.

3. MOTIVATING EXAMPLE

Consider the following very simple loop nest, depending on two parameters N and M ($N > M$), and executing statement `S1`:

```
for (int i=0 ; i<=N ; i++)
   for (int j=0 ; j<=M ; j++)
      S1(i,j);
```

The iteration domain is:

$$\mathcal{P}(N, M) = \{(i, j) \in \mathbb{Z}^2 | 0 \leq i \leq N, 0 \leq j \leq M\}.$$

Imagine that we apply a skewing to this nest, for example to improve locality in the data accesses performed by statement `S1`, or to skew the dependances in order to parallelize the inner loop. The scheduling function is given to CLooG as a scattering matrix representing the transformation: $\begin{pmatrix} i \\ j \end{pmatrix} \mapsto \begin{pmatrix} c1 \\ c2 \end{pmatrix} = \begin{pmatrix} i+j \\ j \end{pmatrix}$.

The resulting polyhedron and CLooG's generated code are depicted Figure 2a and 2c. The presence of `min()` and `max()` in the inner loop induces tests, that generate control overhead, and can be avoided. The key idea of our method is to split the outer loop in several parts, such that each part has single lower and single upper bound, as presented in Figure 2b.

This separation is done by computing the chambers of the inner polyhedra, considering the outer loop indices (including the current one) as parameters. The chambers define constraints on the parameters, and in each chamber there exists an single affine expression of the vertices of the parametric polyhedron [11], or equivalently that the parametric polyhedron has strictly (not piecewise) affine bounds.

In our very simple example, to generate the outermost loops, we take the original domain and consider $c1$ as a parameter. We obtain the polyhedron:

$$\mathcal{I}(N, M, c1) = \{(j) \in \mathbb{Z} | 0 \leq c1 \leq N, 0 \leq j \leq M\}$$

The chambers of this parametric polyhedron (considering the context $N > M$) are:

$$\begin{aligned} \mathcal{C}_1 &= \{(N, M, c1) \in \mathbb{Z}^3 | 0 \leq c1 \leq M\} \\ \mathcal{C}_2 &= \{(N, M, c1) \in \mathbb{Z}^3 | M + 1 \leq c1 \leq N - 1\} \\ \mathcal{C}_3 &= \{(N, M, c1) \in \mathbb{Z}^3 | N \leq c1 \leq N + M\} \end{aligned}$$

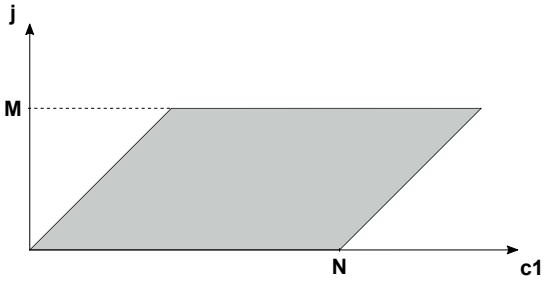
We simply feed CLooG with those three subdomains of the outer loop and it will generate the code given in Figure 2d. This splitting is done just after step 3 of the CLooG algorithm in Figure 1.

These two programs were run on an Intel Xeon CPU E5-2650v3 at 2.30GHz, compiled with gcc 5.4, with the parameter values $N = 10,000,000$ and $M = 1,024$. Statement

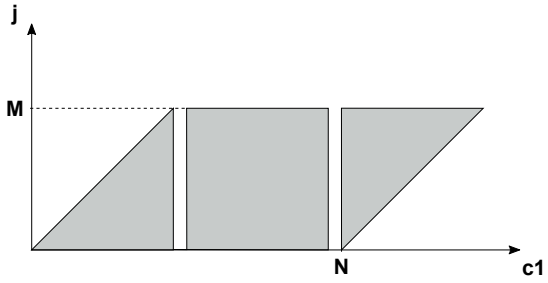
Code Generation:
input: polyhedron list $(\mathcal{T}_{S_1}, ..., \mathcal{T}_{S_n})$, context C (constraints on the parameters and the outer loop bounds), dimension d
output: ordered list of loops, each loop represented as the 3-tuple ($\rightarrow$ P, $\rightarrow$ s, $\rightarrow$ inside), where P is the polyhedron to be scanned in this loop, s the stride of this loop, inside is another ordered list of loops corresponding to the body of this loop

1. intersect all $\mathcal{T}_{S_i}$ with the context C
2. compute all $\mathcal{P}_i$, the projections of those polyhedra onto the outer d dimensions
3. separate the list of $\mathcal{P}_i$ into a list of disjoint polyhedra
4. order them in lexicographical order and build the output list L from them (put them in member P of the list)
5. for each loop D in the list L:
   (a) compute the stride and lower bound for the current loop D at level d, from the inner dimensions of the polyhedra $\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q}$ touched by this loop; update the polyhedron D with respect to the lower bound and store the stride in $D \rightarrow$ s
   (b) create a new list of polyhedra from $\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q}$, merging adjacent polyhedra scanning the same set of statements in this loop subdomain
   (c) $D \rightarrow$ inside = recursively call this function, with arguments: the new list, context $(C \cap (D \rightarrow \text{P}))$, dimension $(d+1)$
6. remove dead code from L by applying steps 2-4 to the polyhedra in $L \rightarrow$ P and removing empty polyhedra
7. merge *point* polyhedra to adjacent *host* polyhedra in L to reduce code size
8. return L

Figure 1: CLooG's algorithm



(a) CLooG domain



(b) Split domain

```
for (c1=0;c1<=N+M;c1++) {
  for (c2=max(0,c1-N);c2<=min(M,c1);c2++) {
    S1((c1-c2),c2);
  }
}
```

(c) CLooG generated code

```
for (c1=0;c1<=M;c1++) {
  for (c2=0;c2<=c1;c2++) {
    S1((c1-c2),c2);
  }
}
for (c1=M+1;c1<=N-1;c1++) {
  for (c2=0;c2<=M;c2++) {
    S1((c1-c2),c2);
  }
}
for (c1=N;c1<=N+M;c1++) {
  for (c2=c1-N;c2<=M;c2++) {
    S1((c1-c2),c2);
  }
}
```

(d) New generated code

Figure 2: Motivating example

S1 performs the assignment of an array by an incrementing counter[2]:

```
A[c1%10][c2] += counter++;
```

The execution times using different compilation options are presented in Table 1. With options `-O3 -march=native`, gcc performs vectorization for all those loops: the CLooG loop and the three split loops. However, it had to insert some runtime tests in the original version while the second split loop does not require any test, resulting in better performance (1.32x).

gcc options	cloog/isl	cloog/split	speedup
-O0	5.72s	5.32s	1.07
-O1	1.33s	0.95s	1.40
-O2	1.44s	0.95s	1.52
-O3 -march=native	0.29s	0.22s	1.32

Table 1: gcc execution times and speedup.

The icc execution times on the same computer and for same code are given in Table 2. It seems that we broke some optimization in icc 17.0 with option `-O1` by splitting the outer loop, since the split version performs more than 2 times worse than the original one! With options `-O2` and `-O3`, icc vectorizes all loop nests in both versions, protecting some of them with tests or peeling some of them. With `-O3`, the speedup reaches 1.67x on this example.

icc options	cloog/isl	cloog/split	speedup
-O0	12.33s	11.95s	1.03
-O1	1.41s	3.38s	0.42
-O2	0.36s	0.25s	1.41
-O3 -march=native	0.27s	0.16s	1.67

Table 2: icc execution times and speedup.

The clang execution times are given in Table 3. Clang version 3.8 with options `-O3 -march=native` did not achieve to vectorize the original CLooG loop nest, but it did vectorize the first and second loop nests of the split version, gaining a very important factor of performance (4.63x).

clang options	cloog/isl	cloog/split	speedup
-O0	7.30s	6.16s	1.18
-O1	1.02s	0.95s	1.08
-O2	1.02s	0.36s	2.85
-O3 -march=native	1.20s	0.26s	4.63

Table 3: clang execution times and speedup.

This first very simple example shows important variations of performance, depending on the compiler, the compilation options, and the polyhedra splitting technique.

[2]We used a modulo 10 in the first array index to avoid allocating a huge array.

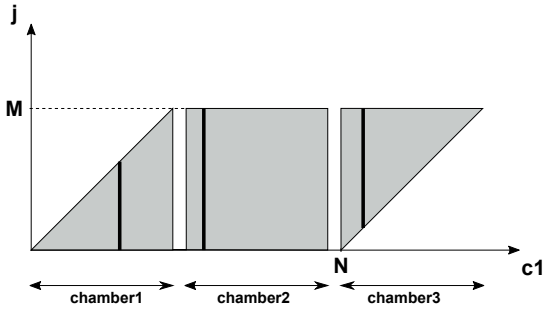


Figure 3: Parametric inner polyhedron

4. SPLITTING POLYHEDRA

Our proposal consists in splitting the domains scanned by a set of successive loops as early as possible, in order to remove all subsequent tests in the inner loops. We will plug our method just after step 3 in CLooG's algorithm (Figure 1): we will split again some of the disjoint polyhedra computed at step 3.

In order to find out how to split a domain, we need to compute its subdomains in which the inner scanned polyhedron is *regular*: its bounds are strictly (not piecewise-) affine functions of the parameters and outer loop indices.

This is the key idea behind this work: we use Loechner and Wilde's algorithm [11] for computing the chambers of a parametric polyhedron, which are exactly those subdomains scanning regular polyhedra. The parametric polyhedron is built from the domain to be scanned, by taking as parameters the program parameters, the outer loop indices and the current loop index. This parametric polyhedron is exactly the inner polyhedron that will be scanned by the inner loops. By computing its chambers, we get the set of domains in the parameters space in which the inner polyhedron is regular. Notice that we do not actually need to compute the parametric vertices, but the complexity of this algorithm resides in the chambers computation.

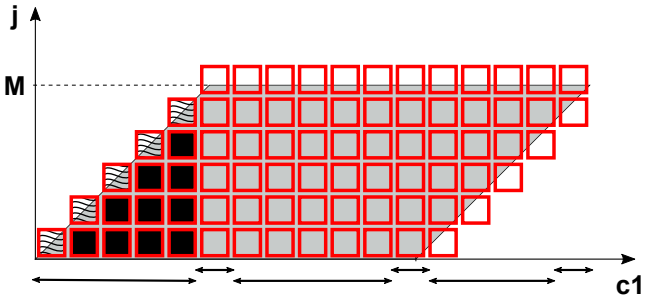
In the motivating example from the previous section, we generate the parametric polyhedron depicted as bold vertical lines in Figure 3 and compute its chambers to split the first loop level.

Let us consider a more complex example: imagine that we tile this domain using 8x8 square tiles. We will show that our technique is also effective to split loops taking out irregular tiles from the regular ones.

The tiled iteration domain and CLooG's standard generated code are given Figure 4. Our algorithm would split those loops in the following way. The original iteration domain is:

$$\mathcal{D}(N,M) = \left\{ \begin{pmatrix} t1 \\ t2 \\ c1 \\ c2 \end{pmatrix} \in \mathbb{Z}^4 \;\middle|\; \begin{array}{l} 0 \le c1 \le N+M, \\ 0 \le c2 \le c1, \\ c1 - N \le c2 \le M, \\ 8*t1 \le c1 < 8*t1+8, \\ 8*t2 \le c2 < 8*t2+8 \end{array} \right\}$$

The first splitting on index $t1$ is done by computing the parametric polyhedron $\mathcal{D}'(N,M,t1) \in \mathbb{Z}^3$ having the same constraints: $t1$ was taken out of the dimensions of this polyhedron and considered as a parameter. There are six chambers in this parametric polyhedron, cutting the domain into six pieces, depicted in Figure 4a as horizontal arrowed lines below the c1 axis.



(a) Tiled iteration domain

```
for (t1=0;t1<=floord(N+M,8);t1++) {
  for (t2=max(0,ceild(8*t1-N-7,8));
       t2<=min(floord(M,8),t1);t2++) {
    for (c1=8*t1;
         c1<=min(min(N+M,8*t1+7),8*t2+N+7);c1++) {
      for (c2=max(8*t2,c1-N);
           c2<=min(min(M,c1),8*t2+7);c2++) {
        S1((c1-c2),c2);
      }
    }
  }
}
```

(b) CLooG generated code

Figure 4: Tiled version

The first chamber is: $\mathcal{C}_1 = \{(t1) \in \mathbb{Z} | 0 \leq t1, 8 * t1 \leq M - 7\}$. Recursing in the algorithm, we generate the $t2$ dimension: the following parametric polyhedron will be scanned in this first chamber:

$$\mathcal{D}(N, M, t1, t2) = \left\{ \begin{pmatrix} c1 \\ c2 \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \begin{array}{l} \text{(same constraints, and)} \\ 0 \leq t1, \\ 8 * t1 \leq M - 7 \end{array} \right\}$$

Computing the chambers of this parametric polyhedron, we obtain as expected two of them, the lower part containing complete tiles (darkened in the figure) and the partial tiles on the upper edge of this triangle (dashed in the figure):

$$\begin{aligned} \mathcal{C}_1 &= \{(N, M, t1, t2) \in \mathbb{Z}^3 | 0 \leq t2, 8 * t2 \leq 8 * t1 - 7\} \\ \mathcal{C}_2 &= \{(N, M, t1, t2) \in \mathbb{Z}^3 | 8 * t1 - 7 < 8 * t2 \leq t1\} \end{aligned}$$

The same process repeats for the five other chambers of the $t1$ dimension. At the end of the algorithm, we obtain 30 different tile shapes being scanned by 30 different loops bodies. The original CLooG version is 9 lines long, it increases in the split version to 205 lines.

In order to limit the complexity —and prevent CLooG from running indefinitely—, we manually limited the number of split domains that are generated to a fixed maximum (256) in the current implementation.

We chose to implement our algorithm in CLooG/PolyLib, since the PolyLib embeds the required chambers decomposition computation, without computing the parametric vertices. We started from the latest CLooG version (0.18.4), with PolyLib low level polyhedral library reactivated (it was no longer maintained since version 0.16 of CLooG). The PolyLib chambers decomposition permits a user to choose between a resulting adjoining or separated set of disjoint chambers. The second choice has to be selected of course, generating loops like $i < lb$ followed by $i \geq lb$. The first, and default choice in PolyLib, would generate $i \leq lb$ as first subdomain and the $i = lb$ iteration would be executed twice.

5. RESULTS

All these benchmarks were run with `CLooG` version 0.18.4, `gcc` version 6.2.1, `icc` version 17.0.0 and `clang` version 3.8.1, in two flavors: without optimizations (`-O0`) and with optimizations (`-O3 -march =native`). The target architecture is a 2.40GHz Intel Xeon E5-2620v3, running linux 4.8. All the programs were run sequentially on a single core. All measurements on CLooG's test suite are an average of 3 runs using the `time` command. The PolyBench measurements were made using the provided script.

5.1 CLooG's test suite

We ran a first set of benchmarks on the many test files that are shipped with the CLooG distribution. Notice that most of them do *not* use tiling. The code generated by our version differs from the mainline CLooG for 46 of them. The resulting code size is usually bigger, with a geometric mean (*geomean*) growth of 257%, and a maximum of 21388% for a corner test case for CLooG. The slowdown of the code generation by CLooG varies from 1x to 100x, with a geomean of 4.5x.

The execution time measurements were performed on those programs, by incrementing a volatile counter in all statements. A summary of the results is presented in Table 4[3]. The first three columns represent the benchmarks that were improved by our splitting method; the three following ones, the benchmarks that were degraded. In each of these groups of three columns, we show the percentage of improved versus degraded benchmarks, the maximum speedup/slowdown, and the geomean speedup. The last column shows the total geomean speedup.

Overall, there are more improvements than degradations. The geomean of speedups for the improved benchmarks is about 10 percent greater than the one of degraded benchmarks. The difference is fading when the compiler optimizations become more agressive: with `-O0` all the control is transfered in the generated code, while it is simplified by the compiler with `-O3`. However, extremal values show that the choice of polyhedral representation is important: from half to twice the performance, depending on the way the polyhedra are split.

On a side note, we noticed that performance and code size growth do not seem to correlate.

5.2 PolyBench

We also ran a test on the PolyBench 4.1 test suite, using the linear-algebra kernels and stencils examples. We ran PLuto 0.11.4 on them, with tiling activated (`-tile` option), and plugged our version of CLooG to generate the output code. The slowdown of the code generation by CLooG varies

[3]The detailed speedup graphs are provided in the appendix.

compiler and option	speedup > 1			speedup < 1			geomean
	percentage	maximum	geomean	percentage	minimum	geomean	speedup
gcc -O0	59	1.34	1.07	41	0.81	0.96	1.02
icc -O0	66.7	1.21	1.03	33.3	0.97	0.99	1.01
clang -O0	82.5	1.96	1.19	17.5	0.5	0.9	1.12
gcc -O3	51.2	1.07	1.03	48.8	0.9	0.97	1.00
icc -O3	61.5	1.11	1.02	38.5	0.89	0.98	1.00
clang -O3	63.9	1.17	1.09	36.1	0.66	0.95	1.03

Table 4: Overview of the speedups for the CLooG examples

compiler and option	speedup > 1			speedup < 1			geomean
	percentage	maximum	geomean	percentage	minimum	geomean	speedup
gcc -O0	80.00	1.13	1.08	20.00	0.93	0.96	1.05
icc -O0	70.00	1.47	1.11	30.00	0.96	0.98	1.07
clang -O0	77.78	1.37	1.15	22.22	0.99	0.99	1.09
gcc -O3	72.73	1.98	1.17	27.27	0.94	0.96	1.11
icc -O3	18.18	1.07	1.04	81.82	0.30	0.79	0.84
clang -O3	66.67	1.25	1.17	33.33	0.88	0.92	1.05

Table 5: Overview of the speedups for the Polybench

from 2x to 50x. The synthetic results are given in Table 5.

Those results confirm our previous observations. With gcc and clang, the split versions are faster on average, up to 1.98x, with an average of 1.11x on all these benchmarks for gcc `-O3`. However, it seems that icc `-O3` performs much better on the original versions than on the transformed ones: the geomean speedup is 0.84x in this case. This is probably due to icc's linear algebra and stencil pattern recognition algorithms performing very agressive optimizations that were ineffective for our transformed versions.

6. RELATED WORK

Minimizing the generated control overhead is a critical challenge for all polyhedral code generation techniques. Two directions coexist to reach that goal: either generating inefficient code then trying to remove its control overhead, or generating directly efficient code.

The first direction started with the seminal polyhedral code generation work by Ancourt and Irigoin [1]. It relies on the Fourier-Motzkin pair-wise elimination technique and generates a significant amount of redundant control as a consequence of Fourier-Motzkin's variable elimination. Their technique was improved by Le Fur [9] by using the simplex method to remove redundant constraints. Kelly et al. showed how to scan several polyhedra in the same code by generating naive perfectly nested loops and then by (partly) eliminating redundant conditionals [8]. Their implementation relies on an extension of the Fourier-Motzkin technique called the Omega test [7]. This technique has been refined later by Chen to minimize control overhead further [5]. While the code generation techniques of this family may also be impacted by considering polyhedra splitting, to the best of our knowledge, this aspect has not been investigated yet to reduce the generated control overhead.

The first technique to generate directly code with a low control overhead has been proposed by Quilleré, Rajopadhye and Wilde [12]. It exploits polyhedral computations to directly generate code free of conditions inside the loops (except some conditions that include modulos), but it may drive to a code explosion. Bastoul proposed changes to this algorithm to limit the code explosion by avoiding splitting polyhedra or fusing back split polyhedra [2, 3]. Vasilache et al. improved it further by mixing the base algorithm with control overhead removal techniques [13]. Grosser et al. added new extensions to, e.g., remove conditions including modulos [6]. They also proposed an *isolation* mechanism to allow the user to specify a part of the space which should be processed separately to, e.g., isolate a vectorizable loop or full tiles. Specifying tiling information to let the code generator extract full tiles is also possible in Reservoir Labs's R-Stream Compiler [4]. As demonstrated in this paper, all techniques based on Quilleré et al.'s algorithm may be strongly impacted by polyhedral splitting. It has been ignored because the code generation tools let the underlying polyhedral libraries choose how to split (or not to split) polyhedra regardless of the code generation problem. Isolation and full tile extraction are a first step towards considering the problem. Our proposal is more general and fully automatic since we are looking for the best splitting in every situation with efficient code generation in mind.

7. CONCLUSION AND PERSPECTIVES

Code generation tools heavily rely on polyhedral operations, usually provided by general-purpose polyhedral libraries that are not designed for code generation. Unions of polyhedra have many possible representations, and depending on the way they are modeled, the generated code can be significantly different.

We proposed a new method to perform a smart splitting to disjoint polyhedra, to generate code without tests or complex bounds in loops, with the objective of reducing control overhead and facilitating vectorization for the compiler. We implemented it in CLooG and compared it to the mainline CLooG on two benchmark suites. Our results show that there can be important performance differences between the generated versions. In some cases, our new technique may significantly improve the quality of the generated code, but in some other cases, it may not be adequate compared to the existing solution. Finding other alternatives and chosing the best one remain open problems to be investigated in

future works.

This paper shows that reaching the best performance is hard, but that polyhedral code generators can attempt to aid the following optimization passes to get closer to the best performance. This leads to a larger reflection on what a polyhedral code generator should feed the compiler with: should it unroll loops? perform hand-tuned vectorization or data prefetching? or should it trust all the optimizing passes that will be applied afterwards?

8. REFERENCES

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[2] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, Slovenia, Oct. 2003.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13, IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[4] C. Bastoul, N. Vasilache, A. Leung, B. Meister, D. Wohlford, and R. Lethin. Extended static control programs as a programming model for accelerators, a case study: Targetting clearspeed csx700 with the r-stream compiler. In *PMEA'09 Workshop on Programming Models for Emerging Architectures*, pages 45–52, Raleigh, North Carolina, Sept. 2009.

[5] C. Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, Beijing, China, 2012. ACM.

[6] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12, 2015.

[7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library. Technical report, University of Maryland, Nov. 1996.

[8] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[9] M. Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency - Practice and Experience*, 8(6):445–460, 1996.

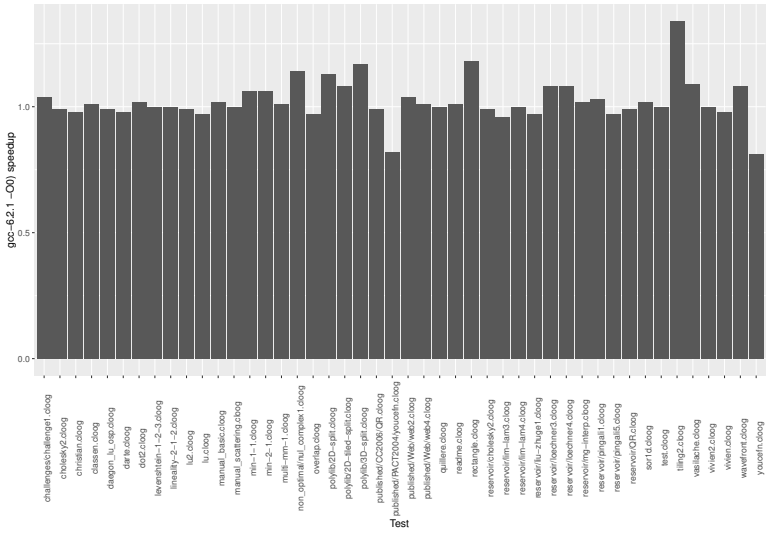
[10] H. Le Verge. Recurrences on lattice polyhedra and their applications, April 1995. Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994.

[11] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, December 1997.

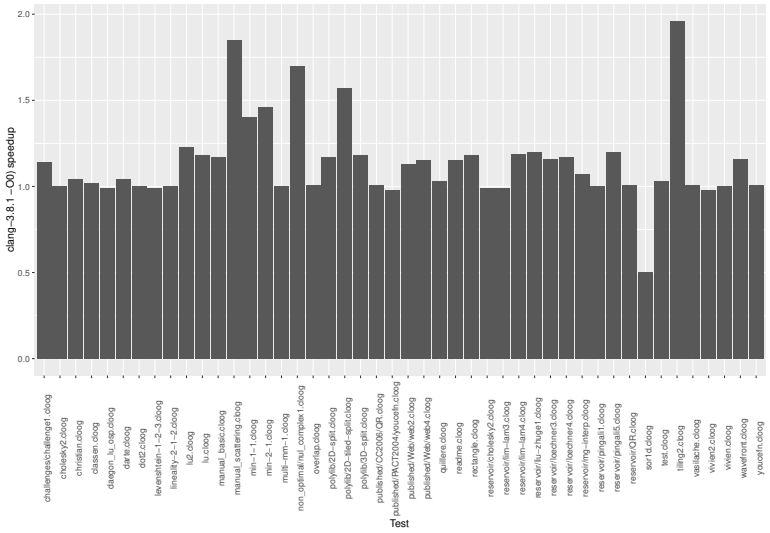
[12] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[13] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, pages 185–201, Vienna, Austria, Mar. 2006.

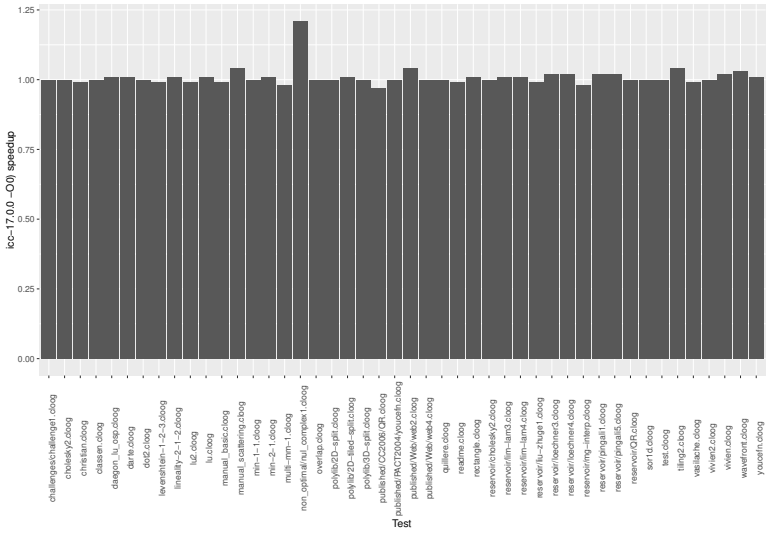
APPENDIX



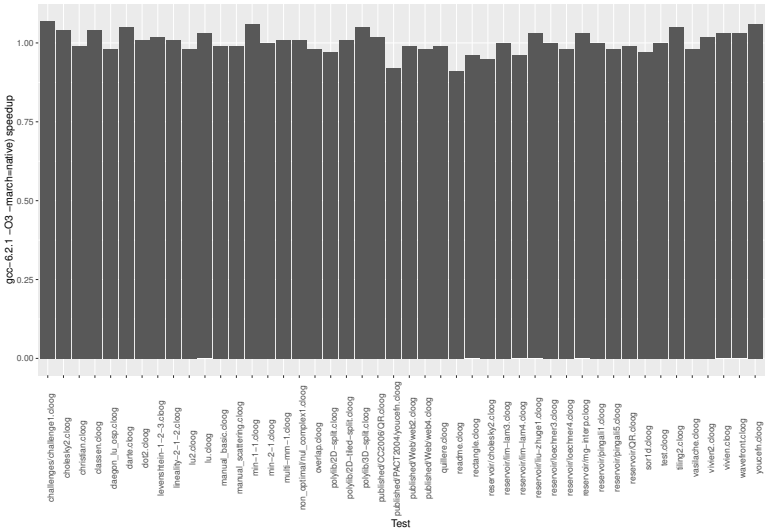
CLooG test suite, gcc -O0



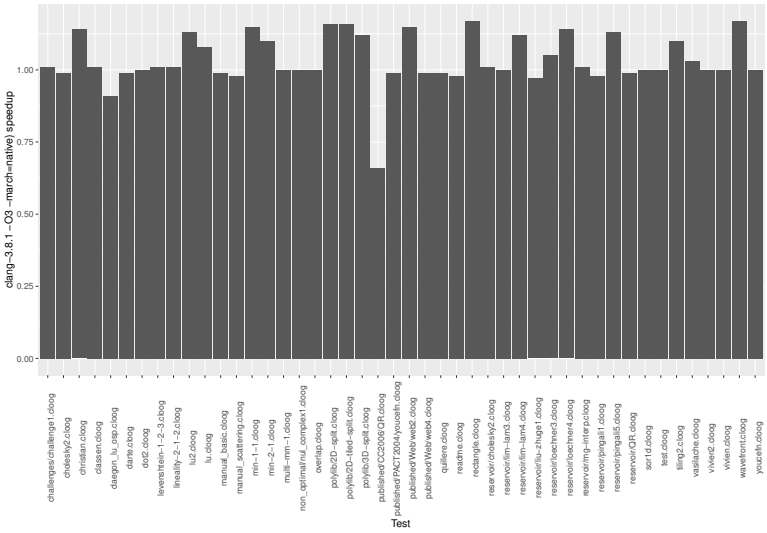
CLooG test suite, clang -O0



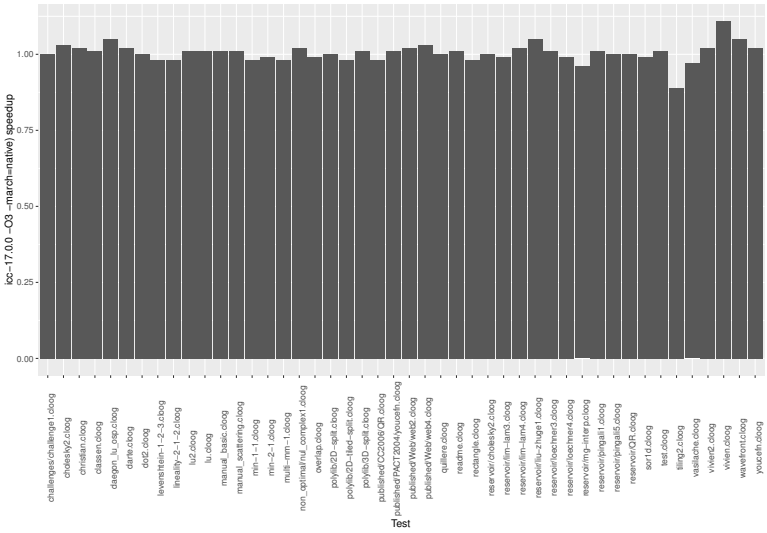
CLooG test suite, icc -O0



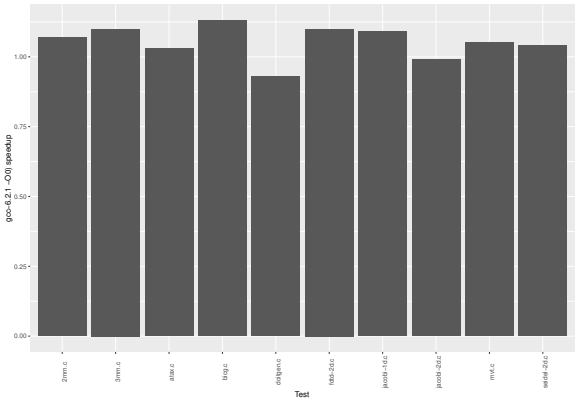
CLooG test suite, gcc -O3



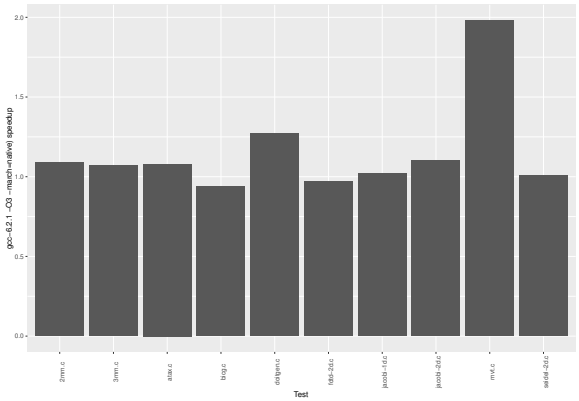
CLooG test suite, clang -O3



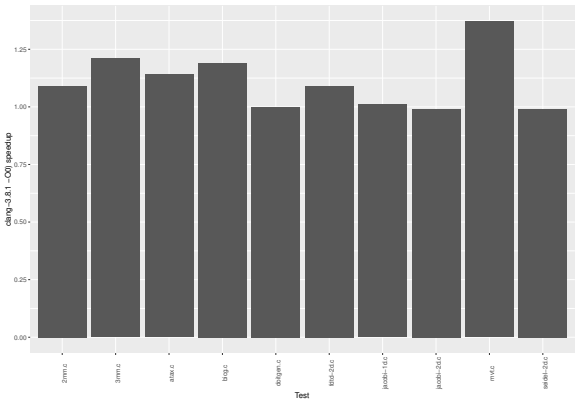
CLooG test suite, icc -O3



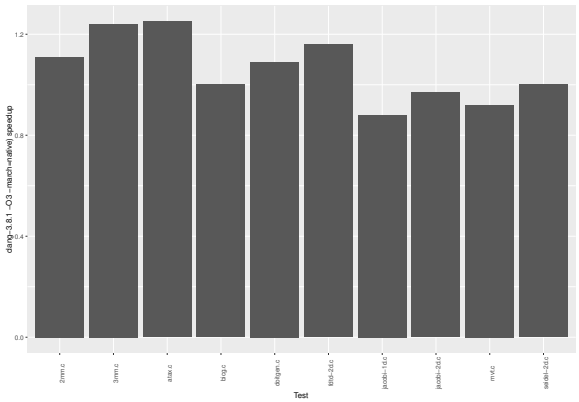
PolyBench, gcc -O0



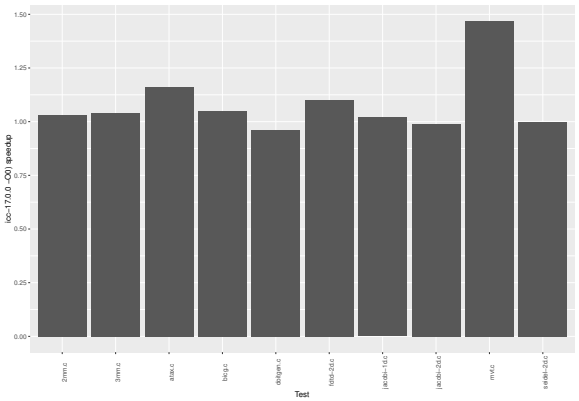
PolyBench, gcc -O3



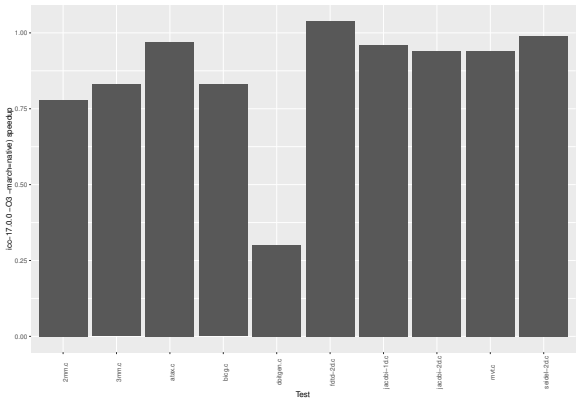
PolyBench, clang -O0



PolyBench, clang -O3



PolyBench, icc -O0



PolyBench, icc -O3